Compression de structure XML pour la recherche d'information structurée

Michel Beigbeder

École Nationale Supérieure des Mines de Saint-Etienne
158, cours Fauriel
F-42023 Saint-Etienne cedex 2
mbeig@emse.fr

RÉSUMÉ. *La recherche d'informations dans les documents structurés nécessite le stockage de la structure des documents indexés dans les index. Si de nombreuses méthodes sont connues et largement utilisées pour compresser les index pour les documents plats, le stockage efficient de la structure est peu étudié. Nous présentons une représentation de structure arborescente adaptée à la recherche d'information structurée, puis nous proposons une méthode de compression des données de cette représentation. Nous présentons les résultats d'expérimentations sur la collection Wikipedia utilisée dans les campagnes INEX 2006 et 2007 (5,8 giga-octet, 659 388 documents) en terme d'efficience en espace et en temps.*

ABSTRACT. *Structured information retrieval needs storing of the document structures in the index. If many methods are known and widely used for the compression of flat document index, the efficient storage of structure has received little attention. We present a structure representation scheme dedicated to structured information retrieval. Then we propose a compression method of this representation scheme. We present experimental results conducted on the INEX 2006 and 2007 Wikipedia collection (5.8 gigabytes, 659 388 documents) both in space and time efficiency.*

MOTS-CLÉS : *Recherche d'information, documents structurés, compression d'index*

KEYWORDS: *Information retrieval, structured documents, index compression*

1. Introduction

La recherche d'information dans des documents structurés prend une importance grandissante. En particulier les structures de documents arborescentes telles que celles que permet le langage XML occupent une place prépondérante. Lors de la phase d'interrogation, certains modèles de recherche d'information ont besoin d'utiliser la structure des documents indexés, en particulier en relation avec la localisation des termes de la requête dans les différents éléments de chaque document pour lequel un score est à calculer. Ces accès peuvent être très nombreux et il faut donc trouver une méthode efficiente pour retrouver la structure du document.

Dans la section 2, nous précisons la définition des besoins et proposons leur résolution par la définition d'une structure de données à associer à chaque élément d'un document XML. Dans la section 3 nous proposons une méthode de compression des fichiers contenant ces structures de données amenant à une diminution de la taille des fichiers de 80%. Dans la section 4 nous montrons que l'efficience s'en trouve améliorée — grâce à la diminution du volume d'entrée-sortie — bien que du temps processeur doive être utilisé pour décompresser les données.

2. Besoin pour la recherche d'information structurée en XML

2.1. *Index positionnel*

Différentes modèles de recherche d'information peuvent être définis sur des documents structurés par une arborescence. Un modèle que nous utilisons [MER 06] est basé sur la proximité des termes de la requête dans les documents et la fonction d'attribution de scores aux documents a donc besoin des positions de toutes les occurrences des différents termes. Pour l'application de ce modèle sur des documents textuels plats, cette information est suffisante. Par exemple, considérons le document suivant où les positions des termes sont indiquées :

$\texttt{Le}_1\ \texttt{joli}_2\ \texttt{titre}_3.\ \texttt{Le}_4\ \texttt{joli}_5\ \texttt{texte}_6\ \texttt{mis}_7\ \texttt{en}_8\ \texttt{emphase}_9.$

Dans un index positionnel, ceci est conservé sous la forme :

$\texttt{emphase} \rightarrow 9$
$\texttt{en} \rightarrow 8$
$\texttt{joli} \rightarrow 2 \rightarrow 5$
$\texttt{le} \rightarrow 1 \rightarrow 4$
$\texttt{mis} \rightarrow 7$
$\texttt{texte} \rightarrow 6$
$\texttt{titre} \rightarrow 3$

Pour appliquer à des documents structurés ce modèle de recherche d'information basé sur les positions des occurrences des termes nous avons besoin pour une occurence donnée de retrouver dans quel nœud elle apparaît. Ce nœud lui-même doit

pouvoir être situé par rapport à son ascendance et ses frères. De ce fait, on doit pouvoir parcourir toute la structure de l'arbre à partir de n'importe quel nœud. Pour le premier besoin, à chaque élément doivent être associées les positions de la première et de la dernière de ses occurrences de termes. Pour le deuxième besoin, la structure arborescente doit être conservée. Par contre nous n'avons pas besoin de conserver les attributs associés à une instance de balise.

2.2. *Arbre XML*

La structure XML est une structure arborescente classique où les éléments d'un même niveau sont ordonnés. Le stockage consiste à conserver pour chaque nœud la liste ordonnée de ses fils. Ceci s'implémente facilement avec deux pointeurs sur chaque nœud : un pointeur sur le premier des fils et un pointeur sur le frère suivant dans le même niveau. Symétriquement, on peut aussi conserver un pointeur sur le dernier des fils et un pointeur sur le frère précédent dans le même niveau. Deux arguments militent en faveur de cette deuxième solution.

D'abord, cette deuxième solution est directement implémentable par numérotation des nœuds lors de leur lecture directe. En effet lors de l'analyse en flot d'un fichier XML, un nœud est entièrement connu lorsque sa balise de fin est rencontrée, avant que ne soit connus les frères qui le suivent, et que soit entièrement connu son père. Par contre, le frère qui le précède est lui déjà connu s'il existe, et s'il n'existe pas c'est que le nœud dont on vient de rencontrer la balise de fermeture est le premier des fils et il se retrouve en queue de liste.

Ensuite, lors de la réutilisation de l'arbre, étant donné un nœud on peut retrouver son numéro au sens *Xpath* en remontant les frères et en comptant les nœuds ayant la même balise. Pour remonter les frères, la liste simplement chaînée avec le frère précédent est donc préférable puisque directement utilisable.

Reprenons le même contenu textuel que celui de l'exemple précédent, cette fois inséré dans une structure XML, et les nœuds y sont donc numérotés selon l'ordre de rencontre de leur balise fermante, ce qui donne la numérotation suivante :

<article>
    <section>
        <titre> Le$_1$ joli$_2$ titre$_3$. </titre>$_{[0]}$
        Le$_4$ joli$_5$ texte$_6$ <emph> mis$_7$ en$_8$ emphase$_9$. </emph>$_{[1]}$
    </section>$_{[2]}$
</article>$_{[3]}$

Les besoins de stockage que nous avons évoqués nécessitent donc de conserver les positions des occurences des termes de début et de fin de chaque élément – deux champs, `start` et `end`, conserveront ces informations –, le pointeur sur le dernier fils (champ `last`) et le pointeur sur le frère précédent (champ `next`). De plus le type de l'élément est stocké par son identifiant de balise (champ `tagid`). Enfin, pour pouvoir

	start	end	last	next	father	tagid
0	1	3	−1	−1	2	2
1	7	9	−1	0	2	138
2	1	9	1	−1	3	6
3	1	9	2	−1	−1	4

Tableau 1. *Le tableau représentant la structure du document exemple.*

remonter dans l'arborescence en O(1), un pointeur sur le père (champ `father`) est également stocké pour chaque nœud.

Une structure permettant de répondre à nos besoins pour un modèle de recherche d'informations utilisant la position des termes dans les éléments de documents structurés est donc composé des champs suivants :

```
struct element {
  WORDPOSITION start;
  WORDPOSITION end;
  TAGIDX last;
  TAGIDX next;
  TAGIDX father;
  TAGID tagid;
};
```

Les types `WORDPOSITION`, `TAGIDX`, et `TAGID` sont des types entiers qui doivent permettre de stocker les valeurs maximales – pour une collection donnée. Le type `WORDPOSITION` est donc lié à la longueur maximale, mesurée en nombre d'occurrences de termes, d'un texte de la collection. Le type `TAGIDX` est lié à la longueur maximale, mesurée en nombre d'éléments, d'un texte de la collection. Enfin, le type `TAGID` est lié au nombre de balises différentes rencontrées dans la collection. Il faut aussi une valeur spéciale pour représenter la valeur NIL des pointeurs et nous utilisons la valeur -1.

Pour ce qui concerne la collection Wikipedia utilisée dans les campagnes INEX de recherche d'information structurée des années 2006 et 2007 [DEN 07], les valeurs maximales sont respectivement : 38 112, 25 057, et 1 263. Avec des types entiers signés il nous faut donc respectivement 4, 2, et 2 octets pour les types `WORDPOSITION`, `TAGIDX`, et `TAGID`. Chaque élément nécessite 16 octets pour son stockage dans la structure.

Les quatre éléments de notre exemple de document structuré se représentent par le tableau 1.

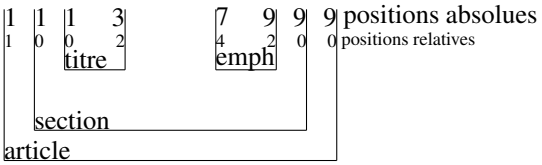

Figure 1. *L'emboîtement des éléments avec leurs positions absolues et relatives.*

3. Méthodes de compression

3.1. *Positions des balises ouvrantes et fermantes des éléments*

L'idée de base des méthodes de compression pour des listes triées de nombres est de stocker les différences entre deux nombres successifs de la liste plutôt que leur valeur. En effet, cette différence est un nombre plus petit que la valeur elle-même, de ce fait elle peut être stockée sur moins de bits. Pour tirer partie de cette propriété, il faut donc mettre en œuvre un codage des entiers sur un nombre variable de bits.

Les plus simples des méthodes pour coder des entiers sur un nombre variable de bits consiste à coder chaque entier sur un nombre entier d'octets. Dans chaque octet, sept bits sont utilisés pour représenter chacun un bit de la mantisse du nombre et le huitième est un bit de continuation qui indique si cet octet est le dernier ou non à considérer. Ainsi les valeurs de 0 à 127 se codent sur un octet, les valeurs de 128 à 16 383 sur deux octets, et plus généralement les valeurs de $2^{7(n-1)}$ à $2^{7n} - 1$ sur n octets.

Pour utiliser cette méthode sur les structures que nous avons définies il faut donc en extraire des listes triées, et que réciproquement ces listes permettent de reconstruire toute l'information présente dans les structures. Les index positionnels manipulent nativement des listes triées : les listes des positions des occurrences. Nous avons aussi des positions dans nos structures. La liste des positions des balises de fermeture des éléments est triée dans le tableau des éléments, cela est dû au fait que les éléments sont numérotés lorsque leur balise de fermeture est rencontrée. Par contre la liste des positions des balises d'ouverture des éléments ne l'est pas. Pour retrouver un ordre sur ces positions, il suffit de remarquer que la position d'ouverture d'un élément est plus petite que la position d'ouverture du premier (dans le document) de ses fils (le dernier dans la liste telle que nous la stockons). De plus on peut définir un ordre sur l'ensemble des positions d'ouverture et de fermeture en tenant compte de toute la structure d'arbre. La position d'ouverture d'un nœud est suivie par les positions d'ouverture puis de fermeture de ses fils en les parcourant dans l'ordre de rencontre dans le document En réutilisant cette propriété récursivement sur tous les nœuds, on en déduit une liste triée de positions. Ceci est illustré par la figure 1 sur notre exemple de document structuré.

Un algorithme qui utilise ces idées doit donc d'abord trouver le premier des frères à un niveau donné, donc le dernier de la liste. Puis en remontant la liste depuis la fin

```
PARCOURS (nœud)

Retourner si nœud est NIL
PARCOURS(frère_précédent(nœud))
IMPRIME(position_de_balise_ouvrante(nœud))
PARCOURS(dernier_fils(nœud))
IMPRIME(position_de_balise_fermante(nœud))
Retourner
```

Figure 2. *Algorithme de parcours des positions par valeur croissante.*

il doit descendre à chaque fois au niveau inférieur. Cela est réalisé avec des appels récursifs.

Lorsqu'un nœud n'a pas de frère précédent, sa position de balise ouvrante est la plus petite des positions de balises non encore traitées. Lorsque tous les fils d'un nœud ont été parcourus, la position de balise fermante du nœud courant est la plus petite des positions de balises non encore traitées. L'algorithme de parcours récursif est sur la figure 2. Sur notre exemple, il imprime la liste de valeurs 1 1 1 3 7 9 9 9. Il suffit ensuite d'insérer dans cet algorithme des calculs de positions relatives par rapport à la dernière valeur traitée pour trouver la liste de valeurs suivantes : 1 0 0 2 4 2 0 0 (cf. figure 1).

3.2. *Pointeurs sur le frère précédent et le dernier des fils*

Les autres entiers utilisés dans les champs de la structure décrivant un élément de la structure sont les pointeurs. Le pointeur sur le père peut être calculé à partir des deux autres. Ces deux autres pointeurs peuvent eux-mêmes être calculés à partir de l'information minimale qui se réduit à savoir si un nœud a des fils et s'il a un frère précédent. Ces deux informations peuvent être stockées chacune sur un seul bit. En effet la numérotation des nœuds qui est faite lors du parcours du document introduit une régularité qui peut être reproduite par un parcours en profondeur en commençant par le dernier des fils (le premier dans la liste stockée). Ainsi l'algorithme de parcours de la figure 3 reconstruit les pointeurs sur le frère précédent et le dernier des fils à partir des deux informations booléennes.

Enfin, pour conserver ces deux valeurs booléennes en utilisant moins d'un octet, elles sont combinées en poids faibles avec les différences de positions utilisées dans

```
NUMÉROTE (nœud, numéro_du_père) retourne le dernier numéro utilisé

Si (nœud a des fils) Alors
    dernier_fils(nœud) = nœud − 1
    dernier_numéro = NUMÉROTE(nœud − 1, nœud)
Sinon
    dernier_fils(nœud) = NIL
    dernier_numéro = nœud
Fin Si
Si (nœud a un frère précédent) Alors
    frère_précédent(nœud) = dernier_numéro − 1
    dernier_numéro = NUMÉROTE(dernier_numéro − 1, numéro_du_père)
Sinon
    frère_précédent(nœud) = NIL
Fin Si
Retourner dernier_numéro
```

Figure 3. *Algorithme de reconstruction des pointeurs sur le frère précédent et le dernier des fils.*

la compression des positions. Pour cela la valeur relative de la position de départ est multipliée par deux et le booléen indiquant la présence d'au moins un fils est noté dans le bit de poids faible (champ `cstart`) ; la valeur relative de la position de fermeture d'un élément est multipliée par deux et le booléen indiquant la présence d'un frère précédent est noté dans le bit de poids faible (champ `cend`). Ceci est illustré sur la figure 4.

Ainsi trois entiers sont stockés pour chaque élément. En plus des deux valeurs combinant position relative et un booléen, il ne faut pas oublier l'identifiant de la balise de l'élément, lequel n'a pas de propriété permettant de le compresser par rapport aux valeurs de son environnement « familial ». Sur notre exemple les valeurs à stocker sont celles des trois dernières colonnes du tableau 2.

4. Mesure d'efficience

Si l'un des gains est effectivement la diminution de l'espace occupé dans les fichiers d'index conservant la structure des documents, la contrepartie est le temps de calcul que le processeur doit effectuer au moment de la compression et de la décompression. Nous avons expérimenté l'efficience de cette méthode de compression aussi

	start	end	last	next	father	tagid	cstart	cend
0	1	3	−1	−1	2	2	0	4
1	7	9	−1	0	2	138	8	5
2	1	9	1	−1	3	6	1	0
3	1	9	2	−1	−1	4	2	0

Tableau 2. *Le tableau représentant la structure du document exemple en version régulière (colonne 1 à 6) et en version compressé (colonnes 6 à 8).*

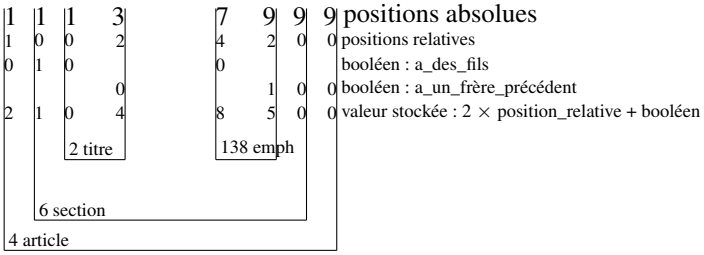

Figure 4. *L'emboîtement des éléments avec leurs positions absolues et relatives, les booléens et les trois valeurs à stocker.*

bien en terme d'espace qu'en terme de temps de calcul. Pour cela nous avons utilisé la collection Wikipedia des campagnes INEX 2006 et 2007 de recherche d'information dans des collections de documents structurés en XML [DEN 07]. Cette collection est composée de 659 388 documents pour un volume de 5,8 giga-octets non compressé (933 méga-octets compressé).

4.1. *Efficience en espace*

Dans la collection Wikipedia, il y a presque 53 millions d'éléments. En conservant les structures des documents avec pour chaque élément une structure de seize octets `struct element` comme indiquée en section 2.2, il faut 52 562 497 × 16 octets, soit environ 802 méga-octets (cf. fichier `struct` dans la table 3). Il faut aussi une donnée complémentaire qui indique les frontières des structures des documents lorsque ces dernières sont toutes concaténées dans un même fichier. Cette donnée a donc autant d'entrées qu'il y a de documents dans la collection plus une, soit 659 389, et avec quatre octets par entrée — un pointeur sur le début de l'ensemble des `struct element` du document considéré — cela fait une taille de 2,5 méga-octets. (cf. fichier `struct.offset` dans la table 3).

Nous avons compressé le fichier des structures avec la méthode décrite ci-dessus et avec l'utilitaire `gzip`. Avec ce dernier, le fichier de structures compressé occupe environ 308 méga-octets, et avec notre méthode le fichier résultant n'occupe que 183 méga-octets. La compression obtenu par notre méthode est bien meilleure, prenant en

taille	taille (octets)	fichier
802 M	840 999 952	`struct`
2.5 M	2 637 556	`struct.offset`
308 M	323 235 277	`struct.gz`
183 M	191 482 677	`cstruct`
48 M	50 838 555	`cstruct.gz`

Tableau 3. *Les tailles des fichiers conservant les structures sous forme native et sous forme compressée.*

1	2	3	4	5	6
57 032 786	20 636 206	21 349 797	17 267 531	4 334 904	2 192 040
7	8	9	10	11	12 ou plus
1 079 042	32 818 599	972 681	3 376	507	22

Tableau 4. *Histogramme du nombre de bits des valeurs à stocker.*

compte les particularités dues aux relations entre les valeurs à conserver. Par contre le fichier obtenu peut encore être fortement compressé par `gzip` (cf. fichier `cstruct.gz` dans la table 3). C'est un indice de possibilité de meilleures performances en efficience en espace.

Pour analyser cette possibilité, nous avons calculé l'histogramme du nombre de bits nécessaires au codage des valeurs qui sont écrites avec un nombre variable d'octets. Cet histogramme est présenté en table 4. On voit la très forte prépondérance des petites valeurs. Une méthode de compression plus forte que celle que nous avons utilisée et qui n'utiliserait pas un nombre entier d'octets permettrait d'obtenir un meilleur taux de compression avec la contrepartie d'un code beaucoup plus lourd et peut-être de temps de calcul un peu plus long.

4.2. *Surcoût de calcul pour la compression*

Pour mesurer le coût de la compression, nous avons réalisé un programme qui pour chaque document de la collection charge la structure de ce document en lisant toutes les structures `struct element` le concernant dans le fichier `struct`, compresse la structure par l'algorithme de PARCOURS (cf. fig. 2) complété avec les calculs de positions relatives, leur combinaison avec les booléens et leur encodage en entiers de taille variable. L'exécution de ce programme utilise environ 28 secondes de temps de calcul en mode utilisateur, 14 secondes en mode système pour un temps total de 1 minute et 30 secondes[1]. Si le code de compression avait été intégré au code d'indexation, seul le temps de processeur en mode utilisateur aurait été à rajouter au temps d'indexation,

1. Toutes les mesures ont été effectuées sur une machine avec processeur Intel à deux cœurs cadencé à 2 GHz et un système d'exploitation Mac OS X, version 10.4.11.

	Première	Deuxième	Troisième	Quatrième
	Temps total			
régulier	1:50.11	0:00.31	0:00.31	0:22.05
compressé	1:27.58	0:00.66	0:00.67	0:35.88
	Temps processeur en mode système			
régulier	1.579s	0.177s	0.179s	1.025s
compressé	1.117s	0.178s	0.181s	1.021s
	Temps processeur en mode utilisateur			
régulier	0.395u	0.125u	0.126u	0.327u
compressé	1.079u	0.479u	0.481u	0.997u

Tableau 5. *Mesure des temps de quatre exécutions de 10 000 accès à des structures de documents en version compressée et régulière.*

en effet le code d'indexation original comprend l'écriture du fichier `struct` qui, étant plus volumineux demande plus de temps système et de temps d'entrée-sortie pour l'écriture effective sur le disque. Comme le temps d'indexation était de 7 minutes et 30 secondes, le surcoût est d'à peine 7 %.

4.3. *Coût d'utilisation des données compressées*

Pour évaluer l'influence de la méthode de compression lors de l'utilisation des données au moment de l'interrogation, nous avons utilisé une stratégie de simulation de besoins d'accès aux structures des documents. Si on considère le fonctionnement du système de recherche d'information dans la phase d'interrogation, après quelques étapes qui peuvent se baser sur des considérations de fréquence des termes qui relèvent des méthodes classiques de la recherche d'information, c'est l'attribution de score à un certain nombre de documents sélectionnés qui va nécessiter d'accéder à leur structure. Comme il n'y a aucune raison qu'il y ait une régularité dans ces besoins, nous simulons ces derniers par une séquence pseudo-aléatoire de numéros de documents. Pour chaque exécution, c'est la même suite de numéros de documents qui est générée. Les mesures ont été effectuées avec une liste de dix mille numéros.

Les entrées-sorties sont déterminantes dans les temps d'exécution. Après démarrage de la machine, quatre exécutions successives ont été effectuées. Lors de la première exécution, toutes les données lues dans les fichiers doivent être effectivement lues sur le disque avec les temps de positionnement et de latence que cela implique. Lors des deuxième et troixième exécutions, toutes les données se trouvent être dans les caches gérés par le système d'exploitation. Pour la quatrième exécution, le mécanisme de cache du système d'exploitation a été désactivé par l'appel système `fcntl()` en positionnant le drapeau `F_NOCACHE`[2]. La table 5 donne les résultats des mesures

2. Cf. la page de manuel de `fcntl(2)`, BSD 4.2.

pour ces quatre exécutions soit avec l'accès à la version régulière (non compressée) des données, soit à leur version compressée selon notre méthode.

Le temps total de première exécution est de 20% meilleur avec la version compressée : moins d'entrée-sortie. Les deuxième et troisième exécution sont deux fois plus rapides avec la version non compressée : toutes les données sont en cache, et il n'y a pas de temps de calcul pour décompresser. Dans la quatrième exécution, les temps entre trois et quatre fois plus petit que dans la première laissent supposer que le disque lui-même intègre un cache.

Les temps d'exécution en mode système sont quasiment les mêmes sauf pour la première exécution où un léger avantage apparaît pour la version compressée.

Quant au temps d'exécution en mode utilisateur, il est naturellement plus grand (entre deux et trois fois) pour la version compressée, mais dans tous les cas négligeable par rapport aux temps des entrées-sorties effectives.

5. Travaux analogues

Clarke *et al.* [CLA 95] proposent d'indexer les balises de documents structurés dans un index positionnel comme les termes ordinaires. De ce fait, ils ne conservent pas l'arborescence des documents, leur modèle de document n'impose d'ailleurs pas que la structure soit arborescente. Par ailleurs, ils proposent une algèbre d'interrogation.

Lee *et al.* [LEE 96] conservent toute la structure arborescente et se préoccupent de l'efficience de différents index. Pour conserver la structure, le principe consiste à encoder les pointeurs de structure dans l'identification des éléments. L'arbre réel du document est projeté dans un arbre k-aire où k est le maximum du nombre de fils d'un nœud ; certains nœuds de ce nouvel arbre sont donc virtuels et ne correspondent pas à des nœuds effectifs. Si par ailleurs, la profondeur de l'arbre du document est n, le plus grand identifiant à manipuler est k^n. Avec $k = n = 10$ – valeurs relativement petites par rapport à ce qu'on peut trouver dans la collection Wikipedia – cela amène à manipuler des nombres de l'ordre de 2^{34}, ce qui nécessite sans compression 34 bits de stockage par identifiant et n'est pas favorable à une efficience en espace. De plus leur proposition a besoin de connaître à l'avance le nombre maximal de nœuds fils de tous les nœuds. Cela impose un premier parcours du document pour fixer k. Leur étude sur l'efficience concerne la comparaison entre la duplication des termes d'indexation d'un nœud dans ses parents en regard de la conservation de la structure sans duplication de l'indexation.

Ces deux méthodes ne prennent pas en compte les positions des termes. Du côté des documents plats (*i.e.* sans structure) de nombreux travaux se sont intéressés à la compression des index. Le livre de Witten *et al.* [WIT 99] fait un tour d'horizon complet sur les méthodes de compression des données et de leur application à la recherche d'information. Il ne traite pas des structures de documents, ni des index positionnels.

L'encodage des entiers par nombre variable d'octets est utilisé depuis les années 80 dans les midifiles [IMA88] pour coder les durées entre les événements. Leur utilisation pour la recherche d'information a été proposé par Williams et Zobel [WIL 96]. Cette méthode de compression est mise en œuvre dans le logiciel ZETTAIR[3]. Les mêmes auteurs ont étudié des méthodes de compression plus performantes pour leur usage en recherche d'information [WIL 99].

6. Conclusion

Nous avons présenté une méthode de stockage des structures pour des documents arborescents XML. Cette méthode permet de retrouver grâce aux positions des termes et des débuts et fin d'élément la localisation dans l'arbre d'éléments de toute occurence d'un terme d'une requête. Nous avons ensuite montré qu'une méthode simple à implémenter permet de compresser avec un facteur de 23% les données initiales de la collection Wikipedia utilisée dans les campagnes INEX 2006 et 2007. Les temps de calcul induits par cette compression sont négligeables à l'indexation et très petits par rapport aux temps d'accès disque à l'interrogation. De plus à l'interrogation, il y a un gain à utiliser des données compressées car cela réduit à la fois le nombre de positionnement sur le disque et la quantité d'octets à transférer sur la liaison disque-ordinateur. Enfin, un dernier avantage est que plus d'informations réside dans les caches avec des données compressées.

On pourrait encore améliorer le taux de compression avec des méthodes plus performantes qui existent. Le coût de décompression serait plus élevé car l'alignement des données que nous avons au niveau octet avec la méthode utilisée serait perdu. Il faudrait étudier si le gain obtenu en espace continuerait d'être intéressant par rapport au surcoût de décompression.

En l'état actuel, le gain le plus important reste le gain en espace sur les fichiers d'index créé, ce qui à matériel donné permet d'augmenter la taille des collections indexées.

Remerciements

Ces travaux sont soutenus par le projet *Web Intelligence* du cluster *ISLE* financé par la région Rhône-Alpes.

7. Bibliographie

[CLA 95] CLARKE C. L. A., CORMACK G. V., BURKOWSKI F. J., « An algebra for structured text search and a framework for its implementation », *The Computer Journal*, vol. 38, n° 1, 1995, p. 43–56.

3. http://www.seg.rmit.edu.au/zettair/

[DEN 07] DENOYER L., GALLINARI P., « The Wikipedia XML corpus », FUHR N., LALMAS M., TROTMAN A., Eds., *Comparative Evaluation of XML Information Retrieval Systems*, n° 4518 Lecture Notes in Computer Science, Springer-Verlag, 2007, p. 12–19.

[IMA88] « Standard MIDI Files 1.0. », rapport, 1988, International MIDI Association.

[LEE 96] LEE Y. K., YOO S.-J., YOON K., BERRA P. B., « Index structures for structured documents », *DL '96 : Proceedings of the first ACM international conference on Digital libraries*, New York, NY, USA, 1996, ACM, p. 91–99.

[MER 06] MERCIER A., BEIGBEDER M., « Calcul de pertinence basée sur la proximité pour la recherche d'information », *Document Numérique*, vol. 9, n° 1, 2006, p. 43–60.

[WIL 96] WILLIAMS H., ZOBEL J., « Indexing nucleotide databases for fast query evaluation », *Proc. International Conference on Advances in Database Technology (EDBT)*, n° 1057 Lecture Notes in Computer Science, Springer-Verlag, March 1996, p. 275-288.

[WIL 99] WILLIAMS H. E., ZOBEL J., « Compressing Integers for Fast File Access », *The Computer Journal*, vol. 42, n° 3, 1999, p. 193–201.

[WIT 99] WITTEN I. H., MOFFAT A., BELL T. C., *Managing Gigabytes. Compressing and Indexing Documents and Images*, Morgan Kaufmann, 1999.